

## I2C LLD SDS

# TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION</b> .....	<b>1</b>
<b>2</b>	<b>KEY FEATURES</b> .....	<b>1</b>
<b>3</b>	<b>HARDWARE SUPPORT (EVM/SOCS)</b> .....	<b>1</b>
<b>4</b>	<b>DESIGN OVERVIEW</b> .....	<b>1</b>
4.1	PERIPHERAL DEVICE DRIVER .....	2
4.2	DEVICE SPECIFIC MODULE LAYER.....	2
4.3	APPLICATION CODE.....	2
4.4	OSAL .....	2
4.5	CSL REGISTER LAYER .....	2
<b>5</b>	<b>MODES OF OPERATION</b> .....	<b>3</b>
5.1	I2C_MODE_BLOCKING.....	3
5.2	I2C_MODE_CALLBACK.....	3
<b>6</b>	<b>DRIVER CONFIGURATION</b> .....	<b>3</b>
6.1	BOARD SPECIFIC CONFIGURATION .....	3
6.2	I2C CONFIGURATION STRUCTURE .....	3
6.3	APIS.....	3
6.4	USAGE.....	4
6.5	API CALLING SEQUENCE.....	4
6.6	FLOW CHART.....	5
<b>7</b>	<b>EXAMPLES</b> .....	<b>7</b>
7.1	EEPROM READ: .....	7
7.1.1	<i>Building the examples:</i> .....	7
7.1.2	<i>Running the examples</i> .....	7
7.1.3	<i>Supported platforms:</i> .....	8
<b>8</b>	<b>TEST</b> .....	<b>8</b>
8.1	BUILDING THE EXAMPLES:.....	8
8.2	RUNNING THE EXAMPLES .....	9
8.3	SUPPORTED PLATFORMS:.....	9
<b>9</b>	<b>MIGRATION GUIDE</b> .....	<b>9</b>
<b>10</b>	<b>BENCHMARKING</b> .....	<b>9</b>

## 1 Introduction

The I2C module provides an interface between a CPU and any I2C-bus-compatible device that connects via the I2C serial bus. External components attached to the I2C bus can serially transmit/receive data to/from the CPU device through the two-wire I2C interface.

## 2 Key Features

- Type of transfers
  - Read
  - Write
  - Write followed by read
- Operating modes
  - Blocking(interrupt or Non interrupt)
  - Callback mode(interrupt)
- Supports only master mode. Slave mode is not supported

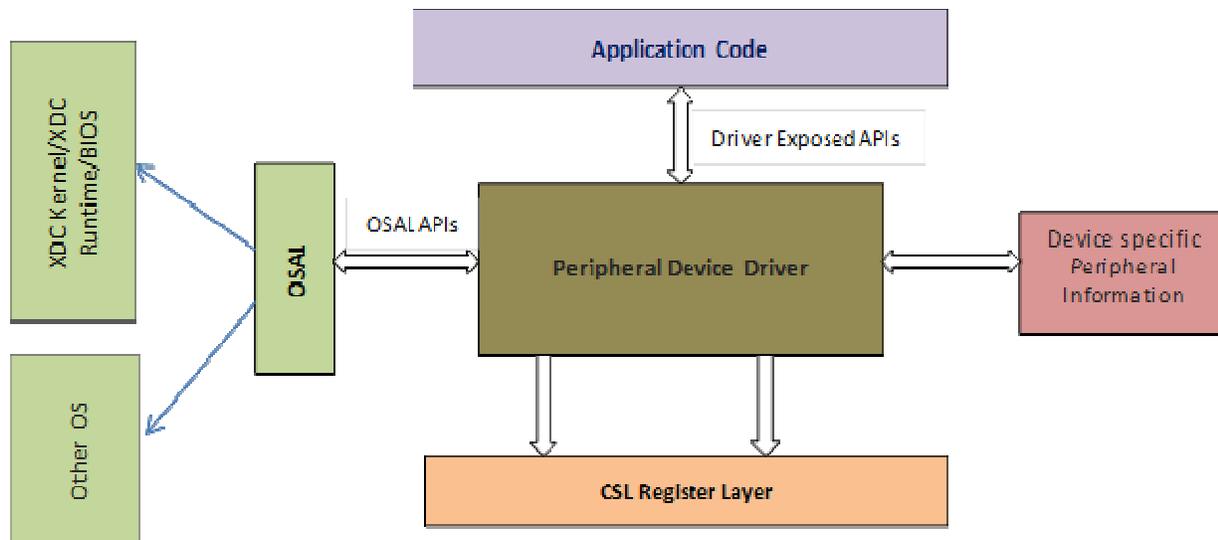
## 3 Hardware Support (EVM/SoCs)

Board	SoC	Cores
AM572x IDK EVM	AM572x	A15 & C66X
AM572x GP EVM	AM572x	A15 & C66X
AM571x IDK EVM	AM571x	A15 & C66X

## 4 Design Overview

The I2C driver provides a well-defined API layer which allows applications to use the I2C peripheral to send and receive data.

The below figure which shows the I2C Driver architecture.



The figure illustrates the following key components:-

#### 4.1 Peripheral device driver

This is the core I2C device driver. The device driver exposes a set of well-defined APIs which are used by the application layer. The driver also exposes a set of well-defined OS abstraction APIs which will ensure that the driver is OS independent and portable. The driver uses the CSL register layer for MMR accesses.

#### 4.2 Device specific module layer

This layer implements a well-defined interface which allows the core I2C device driver to be ported to any device which has the same I2C IP block. This layer may change for every device.

#### 4.3 Application Code

This is the user of the driver and its interface through the well-defined APIs set. Application uses the driver APIs to send and receive data via the I2C peripheral.

#### 4.4 OSAL

The driver is OS independent and exposes all the operating system callouts via this OSAL layer.

#### 4.5 CSL Register Layer

The I2C driver uses the CSL I2C functional layer to program the device IP by accessing the MMR (Memory Mapped Registers).

## 5 Modes of Operation

I2C driver provides the following modes of operations.

### 5.1 I2C\_MODE\_BLOCKING

By default, the I2C driver operates in blocking mode. In blocking mode, a Task's code execution is blocked until an I2C transaction has completed. This ensures that only one I2C transaction operates at a given time.

I2C Driver will support both interrupt or non-interrupt based blocking modes.

### 5.2 I2C\_MODE\_CALLBACK

In callback mode, an I2C transaction functions asynchronously, which means that it does not block a Task's code execution. After an I2C transaction has been completed, the I2C driver calls a user-provided hook function.

Only interrupt based callback mode is supported. Callback mode is not supported in-case of non-interrupt use cases.

## 6 Driver Configuration

### 6.1 Board Specific Configuration

All the board specific configurations like enabling the clock and pin-mux of I2C pins should be performed before calling any of the driver APIs. Once the board specific configuration is done then the driver API `I2C_init()` should be called to initialize the I2C driver.

### 6.2 I2C Configuration Structure

The `I2C_soc.c` file contains the declaration of the `I2C_config` structure. This structure must be provided to the I2C driver. It must be initialized before the `I2C_init()` function is called and cannot be changed afterwards. For details about the individual fields of this structure, see the Doxygen help by opening `\docs\doxygen\html\index.html`.

### 6.3 APIs

In order to use the I2C module APIs, the `I2C.h` header file should be included in an application as follows:

```
#include <ti/drv/i2c/I2C.h>
```

The following are the I2C APIs:

- `I2C_init()` initializes the I2C module.

- **I2C\_Params\_init()** initializes an I2C\_Params data structure. It defaults to Blocking mode.
- **I2C\_open()** initializes a given I2C peripheral.
- **I2C\_close()** deinitializes a given I2C peripheral.
- **I2C\_transfer()** handles the I2C transfer.

## 6.4 Usage

The application needs to supply the following structures in order to set up the framework for the driver:

- **I2C\_Params** specifies the transfer mode and any callback function to be used.
- **I2C\_Transaction** specifies details about a transfer to be performed.
- **I2C\_Callback** specifies a function to be used if you are using callback mode.

## 6.5 API Calling Sequence

The below sequence indicates the calling sequence of I2C driver APIs for a use case of write transaction in blocking mode:

```
I2C_Handle i2c;
UInt peripheralNum = 0; /* Such as I2C0 */
I2C_Params i2cParams;
I2C_Transaction i2cTransaction;
uint8_t writeBuffer[3];
uint8_t readBuffer[2];
bool transferOK;

I2C_Params_init(&i2cParams);
i2cParams.transferMode = I2C_MODE_BLOCKING;
i2cParams.transferCallbackFxn = NULL;

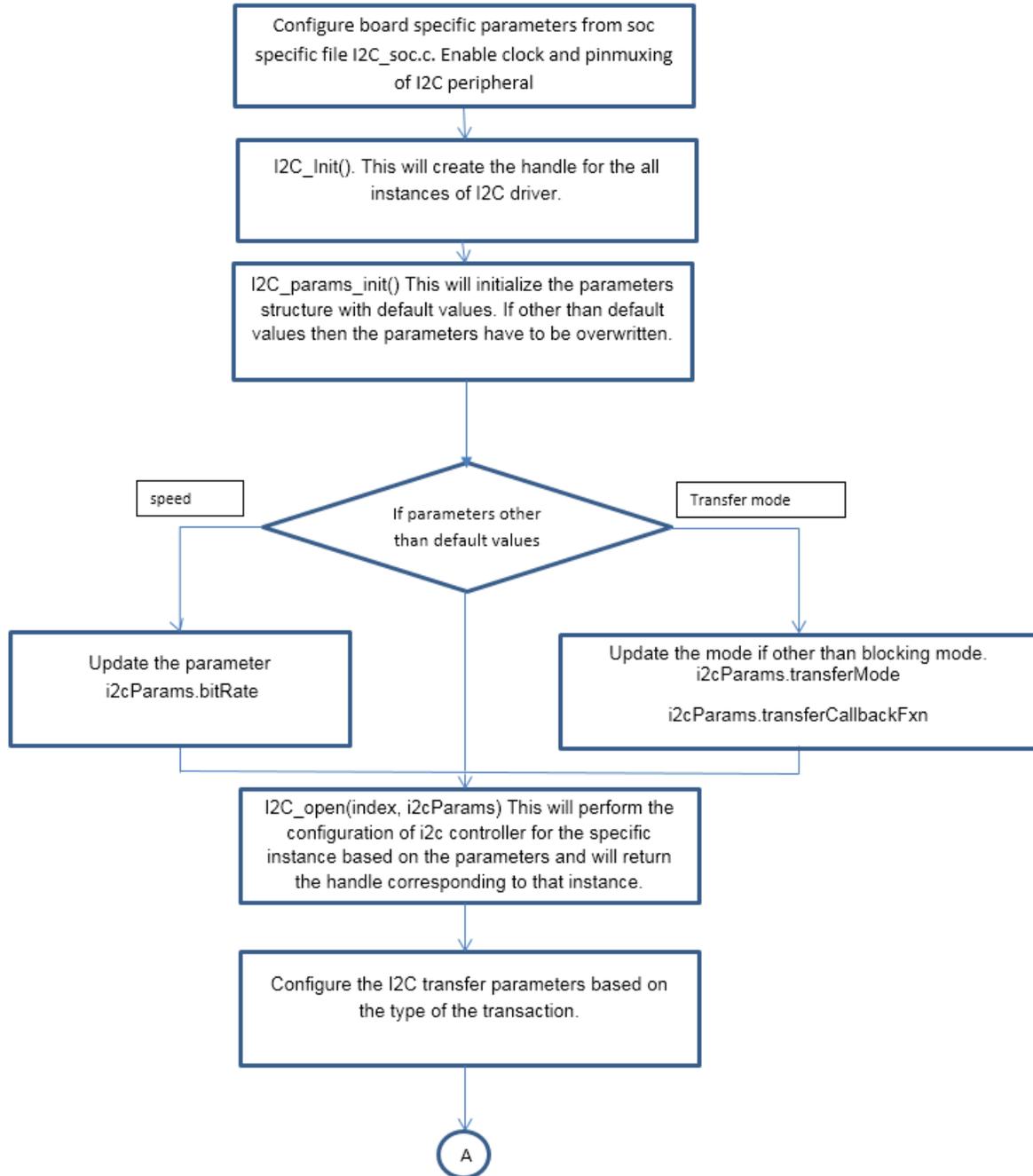
i2c = I2C_open(peripheralNum, &i2cParams);
if (i2c == NULL) {
    /* Error opening I2C */
}

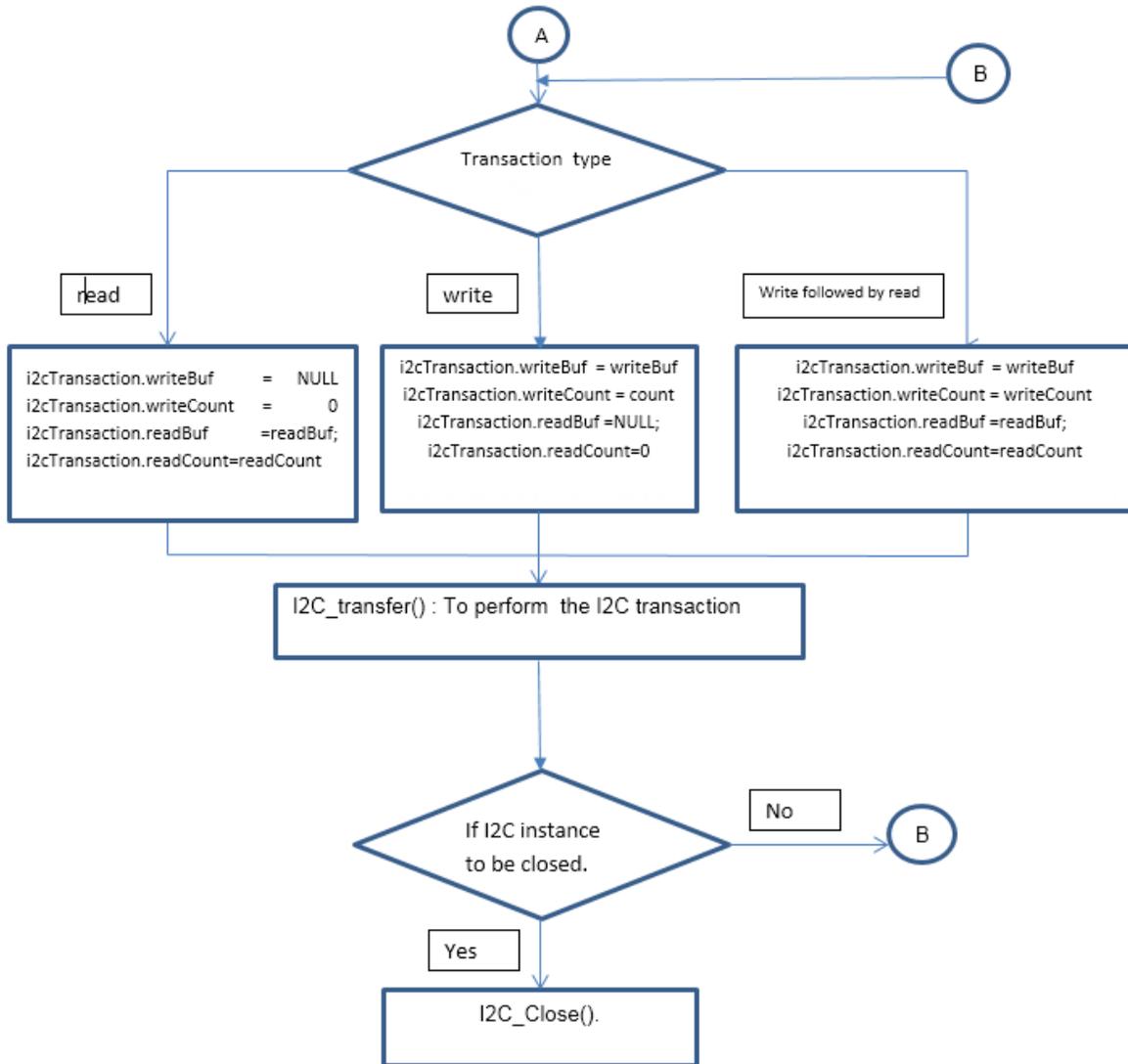
i2cTransaction.slaveAddress = 0x50; /* 7-bit peripheral slave address */
i2cTransaction.writeBuf = writeBuffer; /* Buffer to be written */
i2cTransaction.writeCount = 3; /* Number of bytes to be written */
i2cTransaction.readBuf = NULL; /* Buffer to be read */
i2cTransaction.readCount = 0; /* Number of bytes to be read */

transferOK = I2C_transfer(i2c, &i2cTransaction); /* Perform I2C transfer */
if (!transferOK) {
    /* I2C transaction failed */
}
```

## 6.6 Flow chart

API flow path of I2C driver is as shown in fig.





## 7 Examples

Following are the examples of supported for the I2C Driver

### 7.1 Eeprom Read:

Each EVM will have ID memory (EEPROM), where board specific information like board Id and version information will be stored. This example will read the data from the EEPROM using the I2C driver and verifies whether this data matches with the expected data. As a first step this will write the offset address of EEPROM from which data has to be read and then reads the number of bytes from the EEPROM.

#### 7.1.1 Building the examples:

Following are list of I2C projects which will reside the following location “packages/MyExampleProjects”

```
I2C_BasicExample_AM571X_armExampleProject
I2C_BasicExample_AM571X_c66xExampleProject
I2C_BasicExample_AM572X_armExampleProject
I2C_BasicExample_AM572X_c66xExampleProject
I2C_BasicExample_AM572X_GpEvm_armExampleProject
I2C_BasicExample_AM572X_GpEvm_c66xExampleProject
```

These projects have to be imported in CCS and have to be built. The “.out” files corresponding to each project will be generated after successfully compiling the projects.

Interrupt/Non-Interrupt Modes:

The example projects have to be recompiled to support interrupt and non-interrupt use cases

Following parameter have to be updated in I2C\_Soc.c file and the application have to be recompiled.

- Structure: i2cInitCfg
- Parameter: enableIntr
  - True: interrupt
  - False: Non interrupt

#### 7.1.2 Running the examples

The “.out” have to be loaded and executed. Then on the CCS console the result of the project execution will be displayed. If the project is executed successfully, then it will print “PASS” else will print “Data Mismatch”

### 7.1.3 Supported platforms:

AM572x GP EVM  
AM572x IDK EVM  
AM571x IDK EVM

## 8 Test

Each EVM will have ID memory (EEPROM), where board specific information like board Id and version information will be stored. This example will read the data from the EEPROM using the I2C driver and verifies whether this data matches with the expected data. As a first step this will write the offset address of EEPROM from which data has to be read and then reads the number of bytes from the EEPROM.

This test application will test the use case for the following speeds  
100 Kbps  
400 Kbps

### 8.1 Building the examples:

Following are list of I2C projects which will reside the following location  
“packages/MyExampleProjects”

I2C\_BasicExample\_AM571X\_armTestProject  
I2C\_BasicExample\_AM571X\_c66xTestProject  
I2C\_BasicExample\_AM572X\_armTestProject  
I2C\_BasicExample\_AM572X\_c66xTestProject  
I2C\_BasicExample\_AM572X\_GpEvm\_armTestProject  
I2C\_BasicExample\_AM572X\_GpEvm\_c66xTestProject

These projects have to be imported in CCS and have to be built. The “.out” files corresponding to each project will be generated after successfully compiling the projects.

Interrupt/Non-Interrupt Modes:

The example projects have to be recompiled to support interrupt and non-interrupt use cases

Following parameter have to be updated in I2C\_Soc.c file and the application have to be recompiled.

- Structure: i2cInitCfg

- Parameter: enableIntr
  - True: interrupt
  - False: Non interrupt

## 8.2 Running the examples

The “.out” have to be loaded and executed. Then on the CCS console the result of the project execution will be displayed. If the project is executed successfully, then it will print “PASS” else will print “Data Mismatch”

## 8.3 Supported platforms:

AM572x GP EVM  
AM572x IDK EVM  
AM571x IDK EVM

## 9 Migration Guide

The driver supports multiple SoCs, Cores and different IP versions. Different IP versions are supported using function pointer based approach, whereas high lever driver APIs will remain same and these driver APIs will call the corresponding the correct version of IP specific implementation APIs using function pointers. This function pointer table will be fixed for each instance of the peripheral and will be defined in the main config structure, which resides in soc specific file “I2C\_soc.c”.

Users who are using the low level APIs(Device abstraction APIs: which perform hardware register read/write) have to use the high level APIs which are described in the section 6.3.

## 10 Benchmarking

Code size for library in bytes:

Initialized data : 40  
Code: 5792